Online Algorithms

Kim Skak Larsen

Department of Mathematics and Computer Science
University of Southern Denmark

kslarsen@imada.sdu.dk

Advanced Algorithms (DM582)
April 13, 2026

About These Slides

Disclaimer

These slides contain much more text than I usually put on slides.

The reason is that no good text exists for this material at this level. So, the slides should replace a textbook.

Thus, the slides will be less suited for lecturing.

Online Problems and Algorithms

A problem is an *online* if the following holds:

- We must process a non-empty, finite request sequence.
- Each request must be processed before we receive the next request.
- We must make an irrevocable decision for each request.
- The decisions while processing the sequence results in some cost.

In the intro course, you saw

- ski rental [1]
- bin packing
- machine scheduling

An *online algorithm* is an algorithm for an online problem.

[1] A simplified version of power saving for screens, etc.

Competitive Analysis

Competitive analysis is one way to measure the quality of an online algorithm, i.e., how are we at keeping the cost low.

Idea

Let Opt denote an *optimal offline algorithm.*

"Offline" means getting the entire input before having to compute, which corresponds to knowing the future!

We calculate how well we perform compared to Opt.

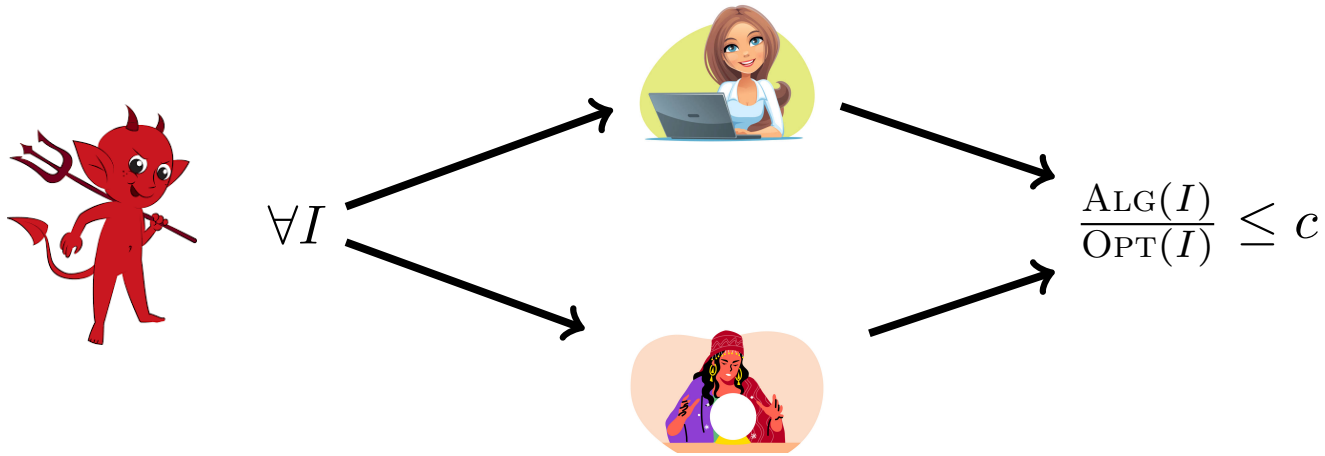
Notation

$\textsc{Alg}(I)$ denotes the cost of running algorithm Alg on the request sequence I.

Thus, $\textsc{Opt}(I)$ is the cost of running Opt on I.

We are intuitively interested in minimizing $\frac{\textsc{Alg}(I)}{\textsc{Opt}(I)}$, but this of course varies with I. To give a guarantee, we must find the ratio for a worst sequence.

Competitive Analysis



$\forall I$ is one way to look at it, but we often talk about it as a game between us and an *adversary* that knows our algorithm and tries to make the hardest possible sequence for us.

An algorithm, Alg, is *c-competitive* if there exists a constant, b, so that

$$\forall I: \ \text{Alg}(I) \leq c\,\text{Opt}(I) + b$$

Alg has *competitive ratio* c if

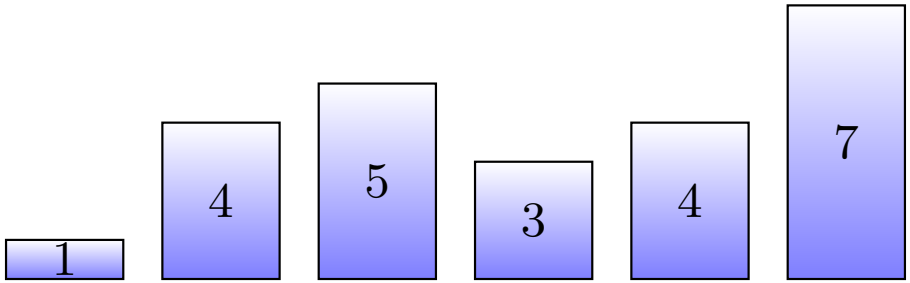
c is the *best*[2] (smallest) c for which Alg is c-competitive.

[2] Formally, it is $\inf\{c \mid \text{Alg is } c\text{-competitive}\}$.

Review of Machine Scheduling

- $m \geq 1$ machines.
- n jobs of varying sizes arriving one at a time to be assigned to a machine.
- The goal is to *minimize makespan*, i.e., finish all jobs as early as possible.
- Algorithm List Scheduling (Ls): place next job on a least loaded machine.

Machine Scheduling: List Scheduling example



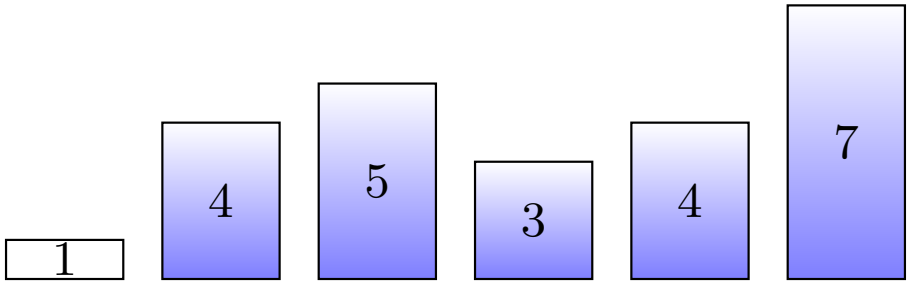
M_1 M_2 M_3 M_4

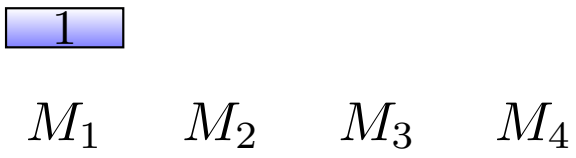
LS

M_1 M_2 M_3 M_4

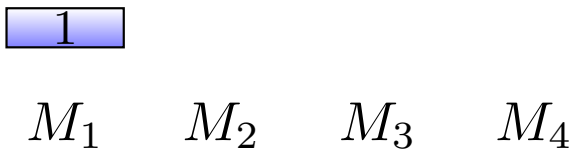
OPT

Machine Scheduling: List Scheduling example



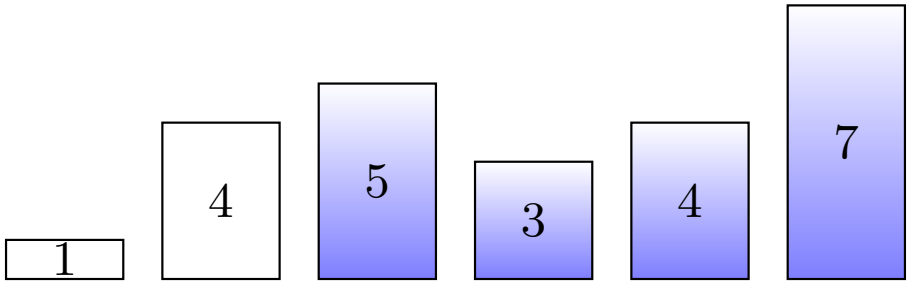


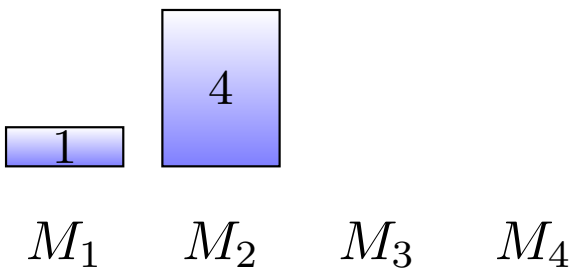
LS



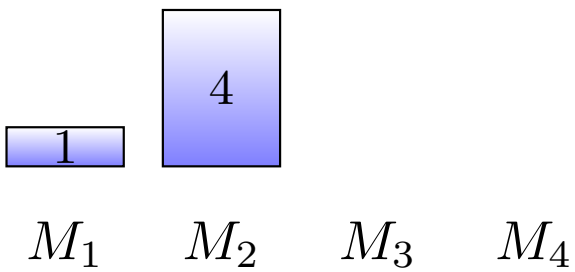
OPT

Machine Scheduling: List Scheduling example



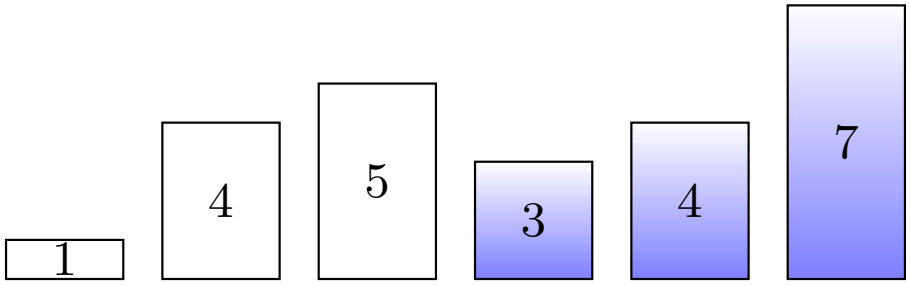


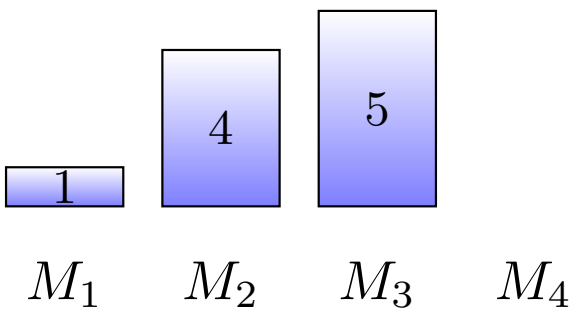
LS



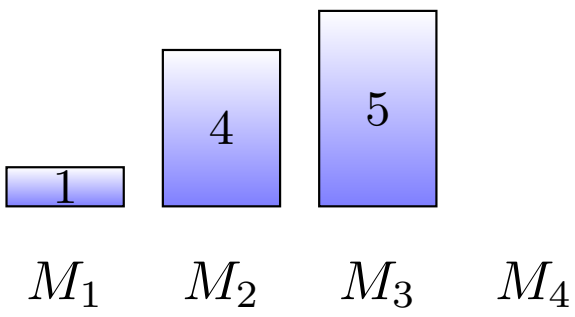
OPT

Machine Scheduling: List Scheduling example



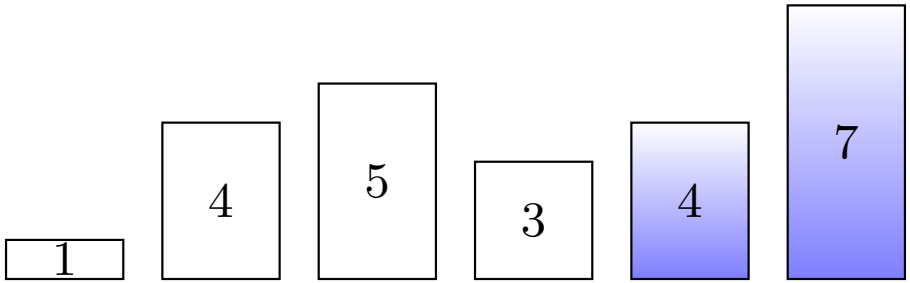


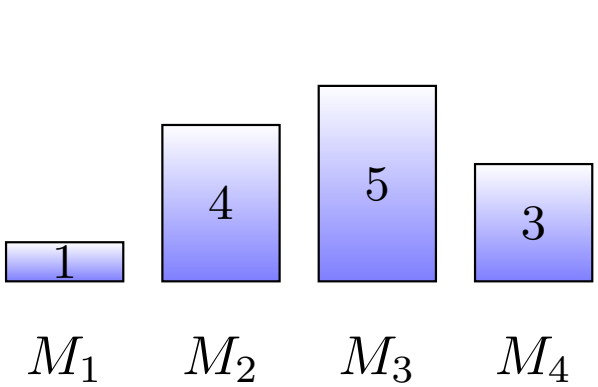
LS



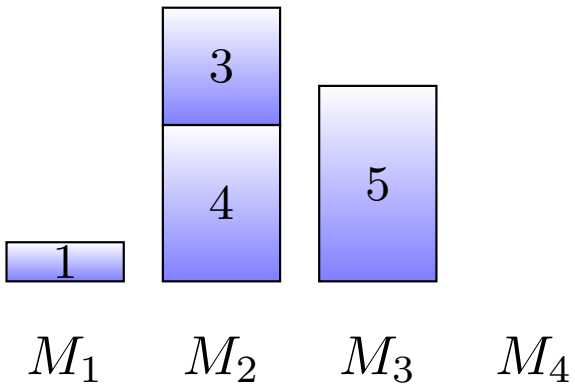
OPT

Machine Scheduling: List Scheduling example



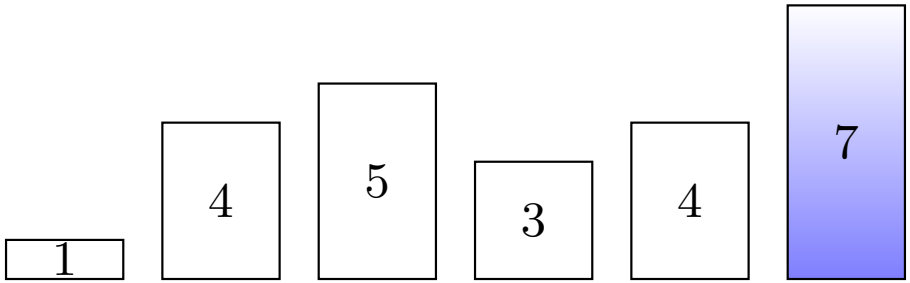


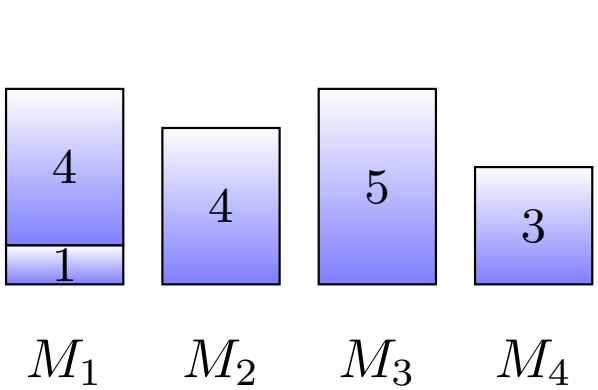
LS



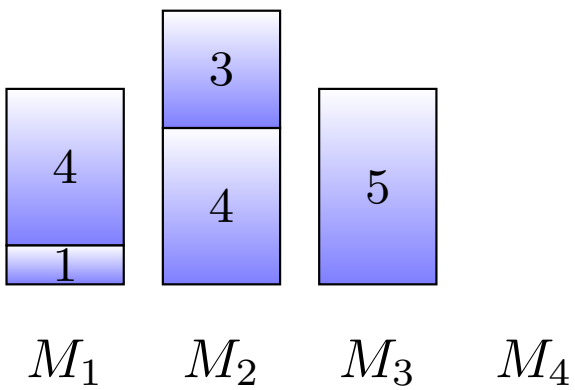
OPT

Machine Scheduling: List Scheduling example



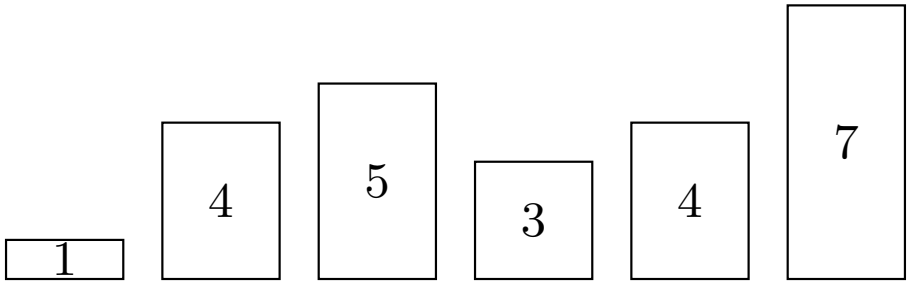


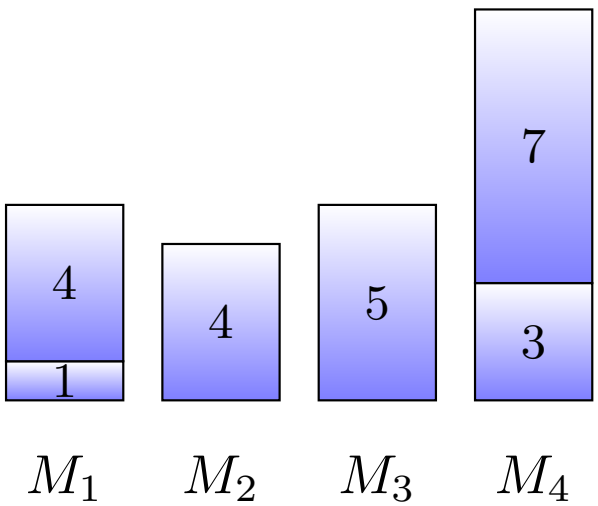
LS



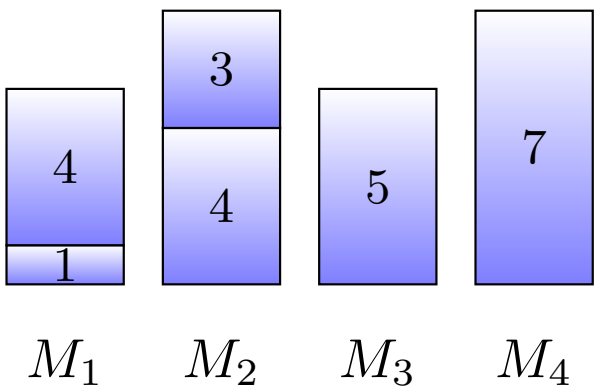
OPT

Machine Scheduling: List Scheduling example



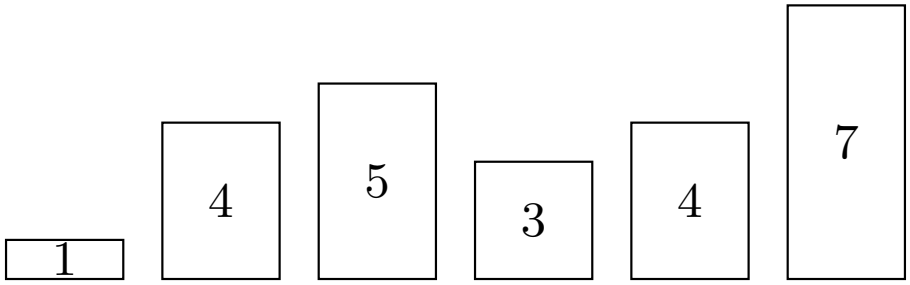


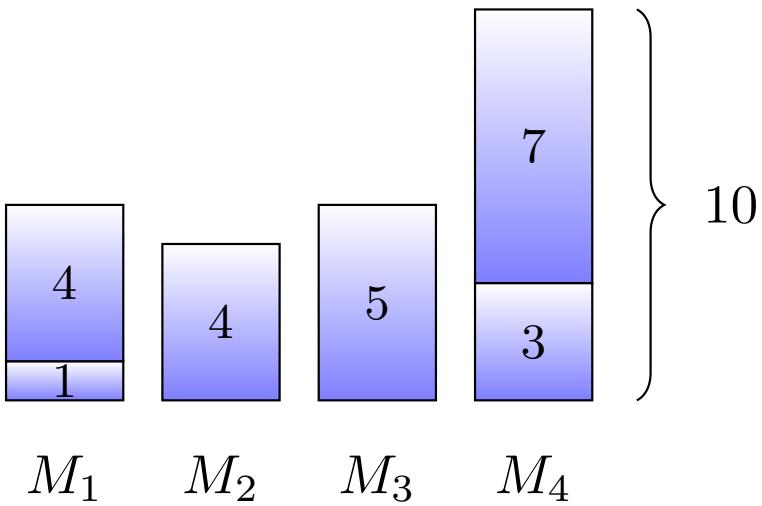
LS



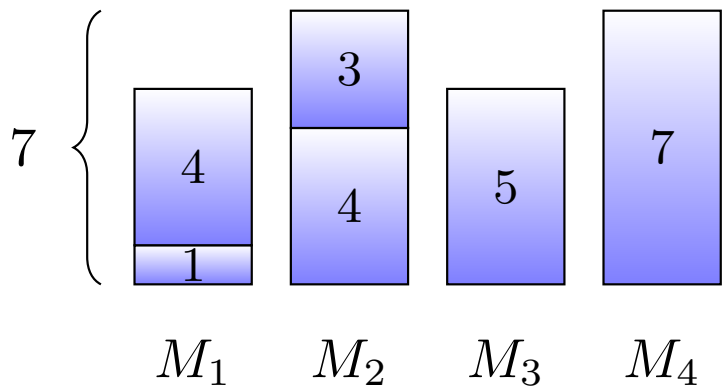
OPT

Machine Scheduling: List Scheduling example



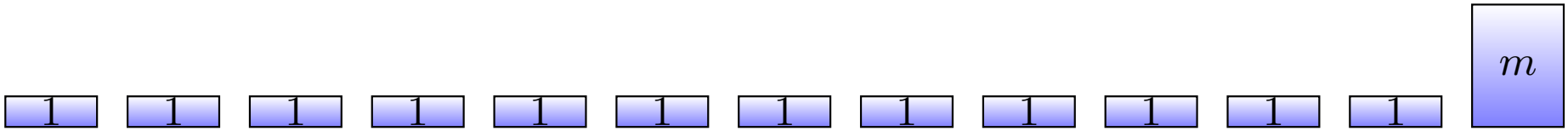


LS



OPT

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



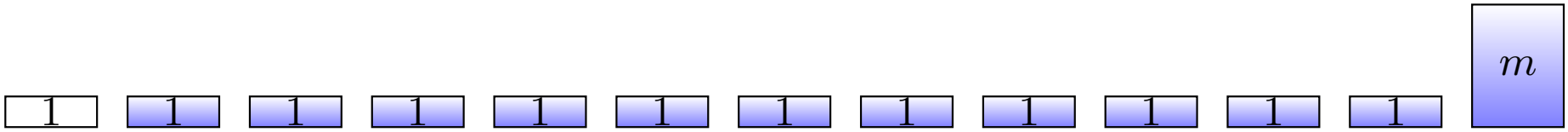
M_1 M_2 $\cdots$ M_m

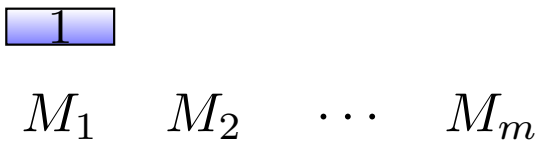
Ls

M_1 M_2 $\cdots$ M_m

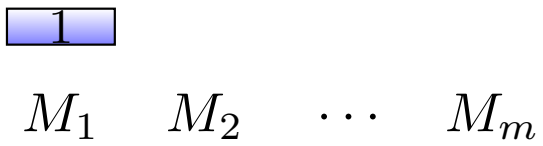
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



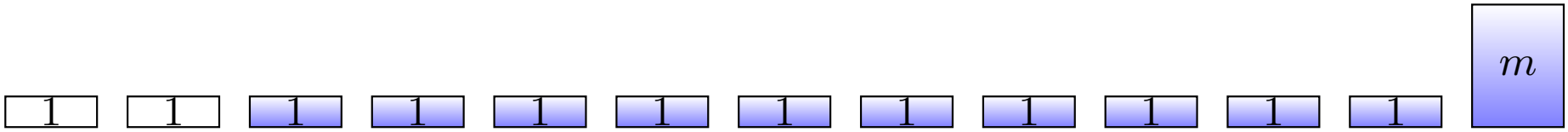


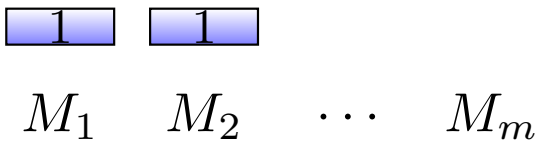
Ls



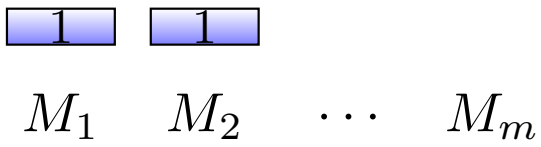
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



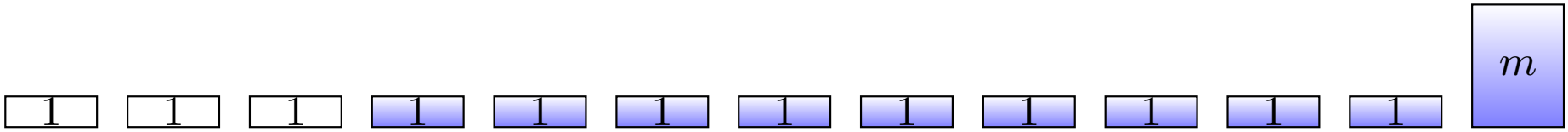


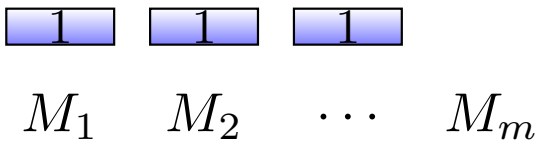
Ls



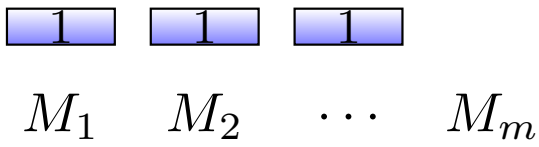
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



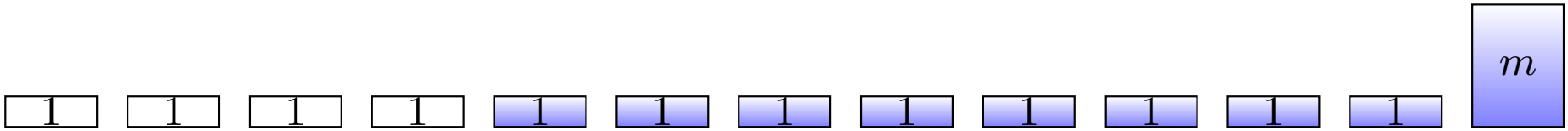


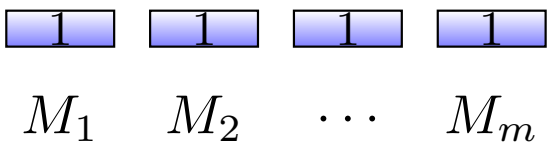
Ls



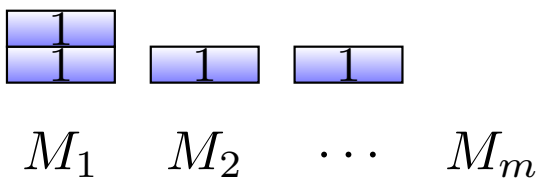
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



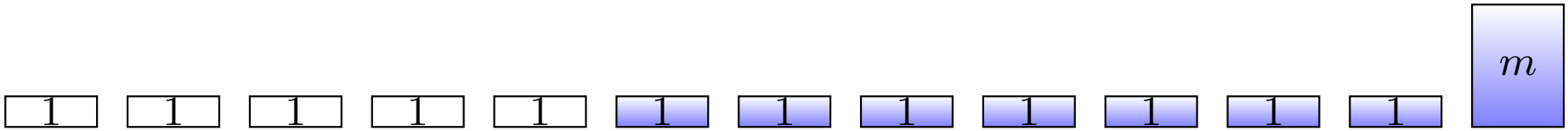


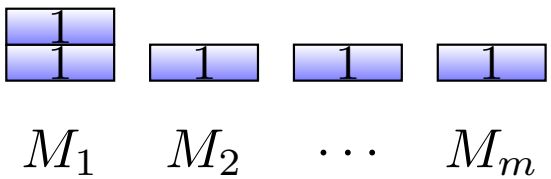
Ls



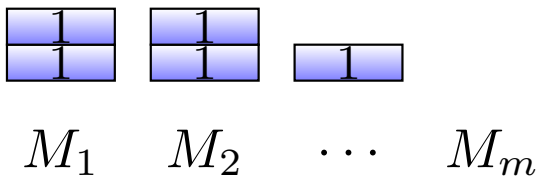
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



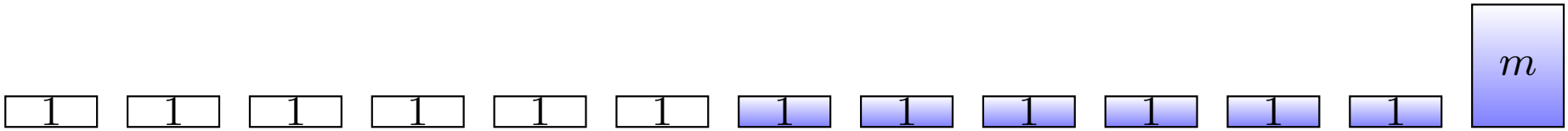


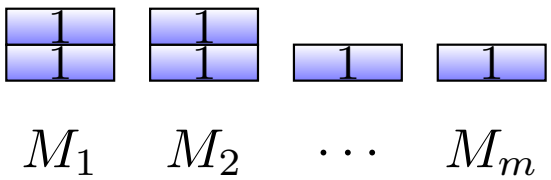
Ls



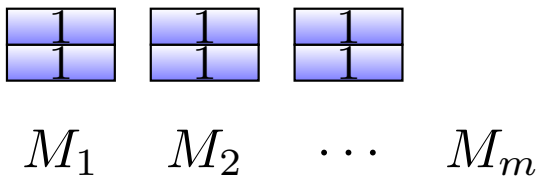
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



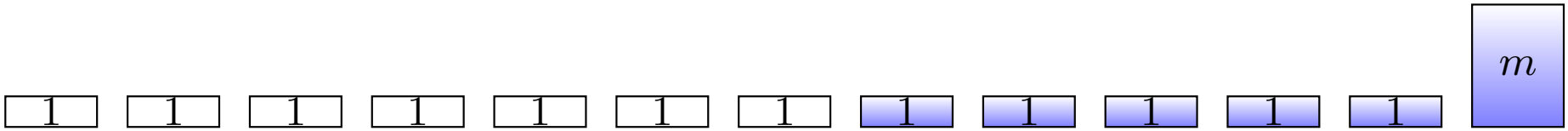


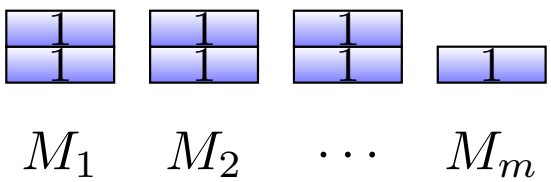
Ls



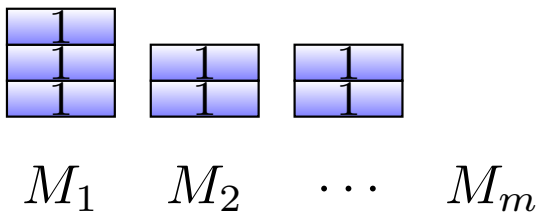
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



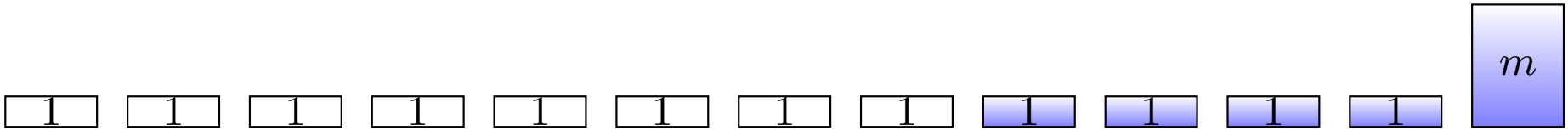


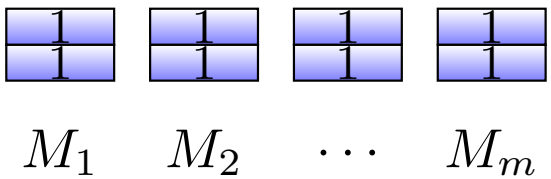
Ls



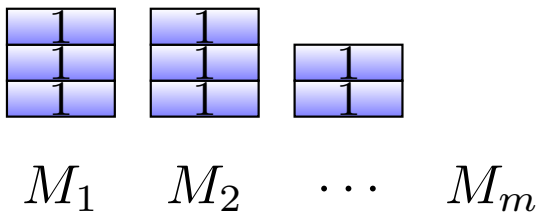
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



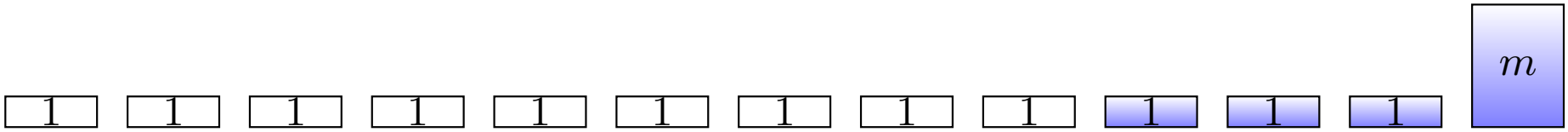


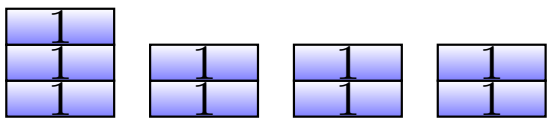
Ls



Opt

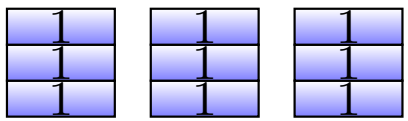
Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive





M_1 M_2 $\cdots$ M_m

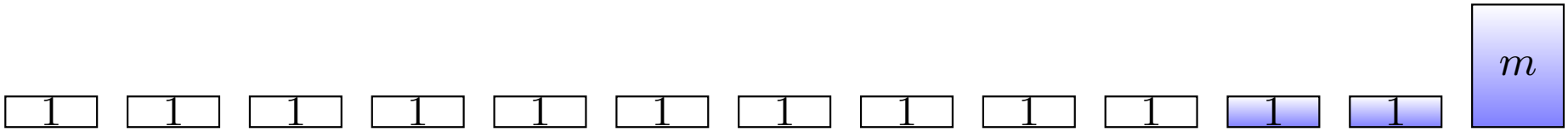
Ls

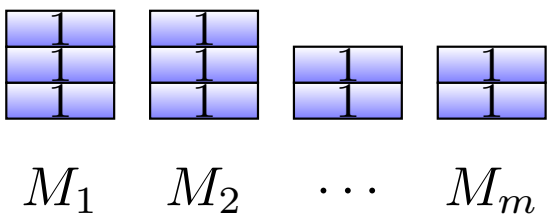


M_1 M_2 $\cdots$ M_m

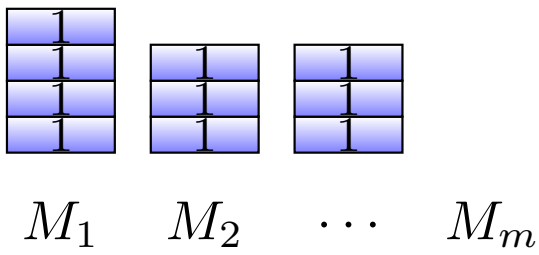
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



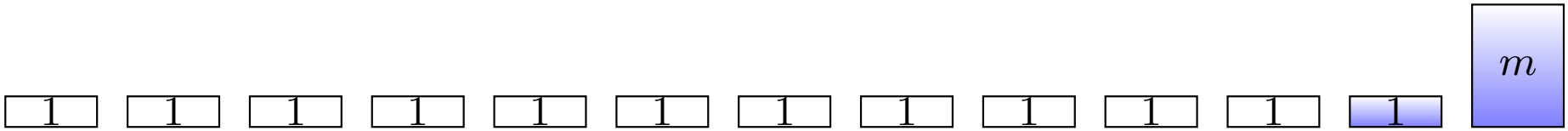


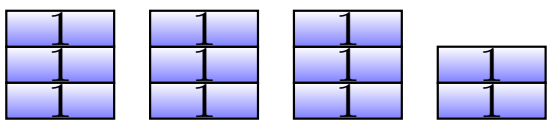
Ls



Opt

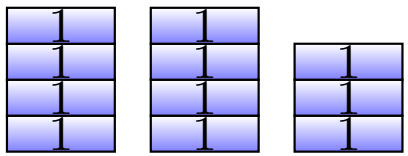
Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive





M_1 M_2 $\cdots$ M_m

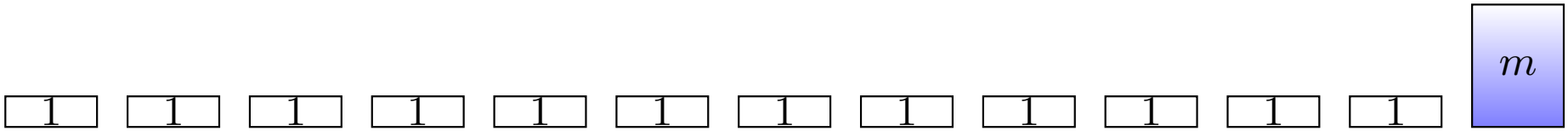
Ls

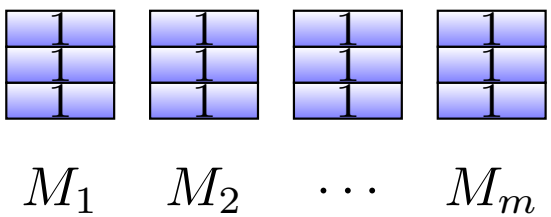


M_1 M_2 $\cdots$ M_m

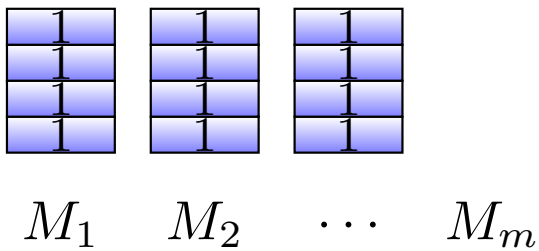
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



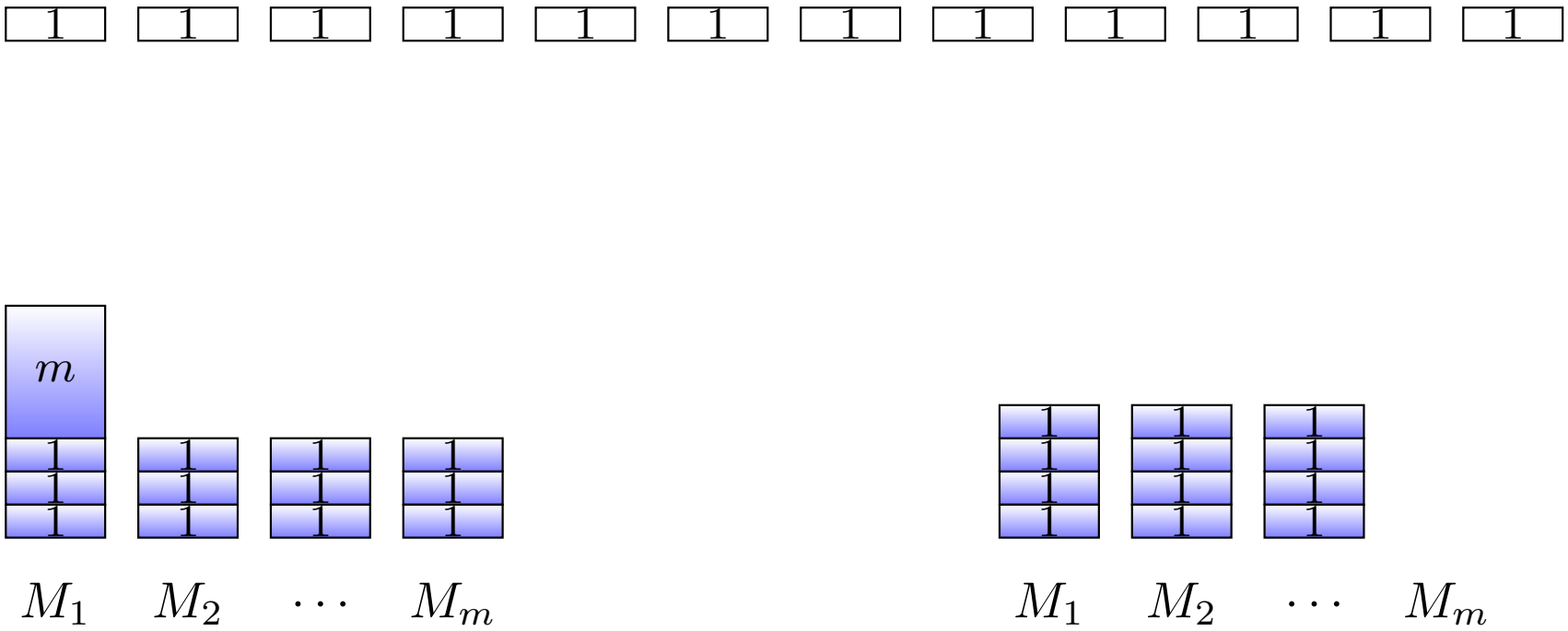


Ls

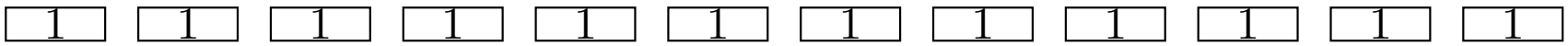


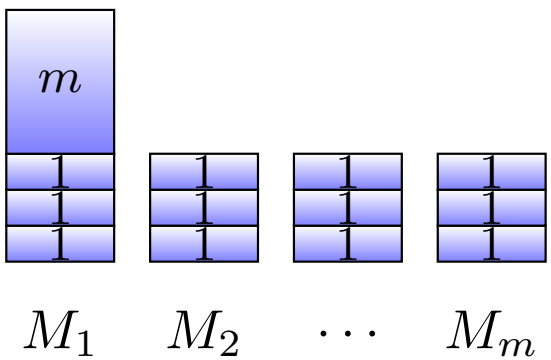
Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive

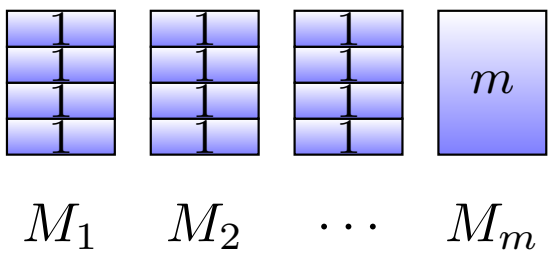


Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive



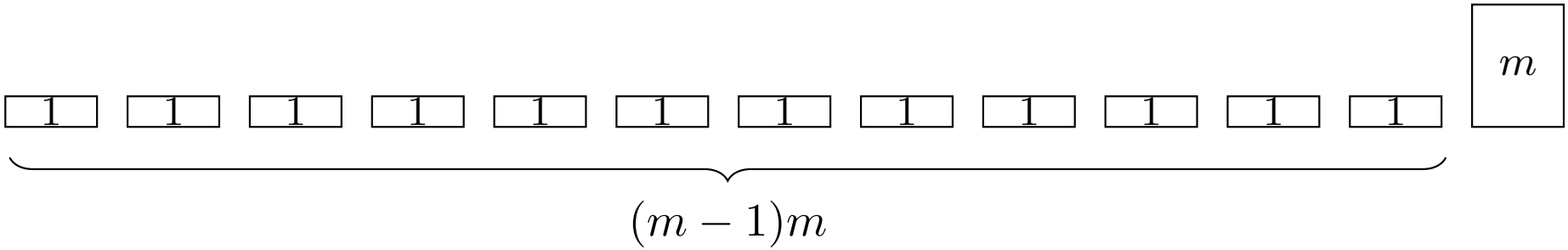


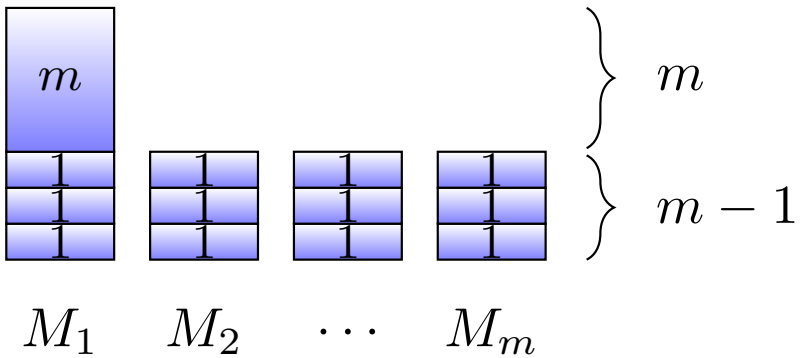
Ls

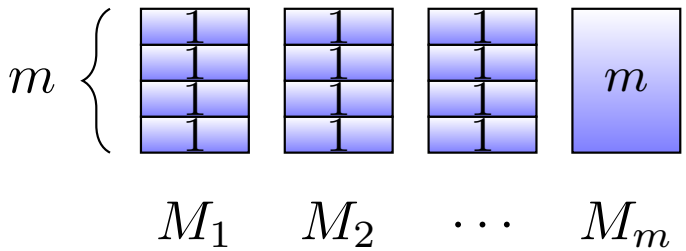


Opt

Machine Scheduling: Ls is at best $(2 - \frac{1}{m})$-competitive

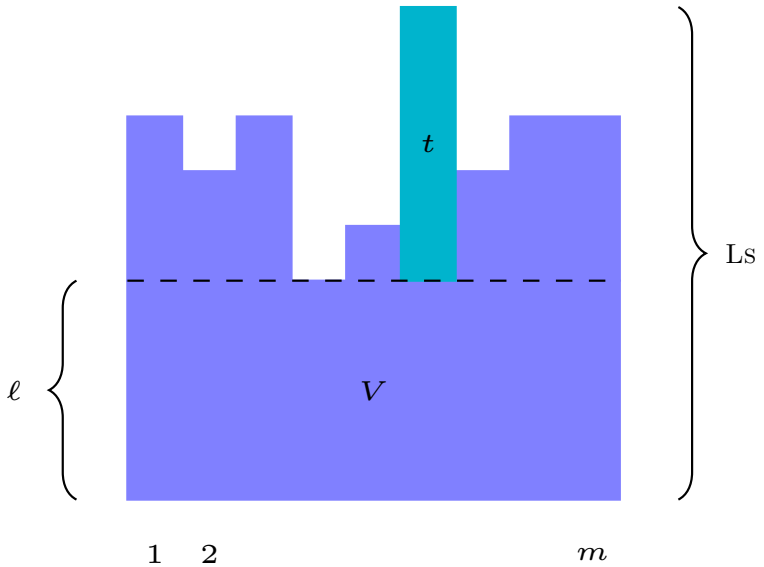






$$\frac{\text{Ls}(I)}{\text{Opt}(I)} \geq \frac{(m-1)+m}{m} = \frac{2m-1}{m} = 2 - \frac{1}{m}$$

Machine Scheduling: Ls is $(2 - \frac{1}{m})$-competitive



t is the length of the job starting at ℓ and ending at the makespan

T is the total length of all jobs

Define $V = \ell \cdot m$

Now conclude:

$\textsc{Opt} \geq T/m$ and $\textsc{Opt} \geq t$

$T \geq V + t$, due to Ls's choice for t

$$
\begin{aligned}
\textsc{Ls} &= \ell + t \\
&\leq \tfrac{T-t}{m} + t, \text{ since } \ell = \tfrac{V}{m} \text{ and } V \leq T - t \\
&= \tfrac{T}{m} + (1 - \tfrac{1}{m})t \\
&\leq \textsc{Opt} + (1 - \tfrac{1}{m})\,\textsc{Opt}, \text{ from } \textsc{Opt} \text{ inequalities above} \\
&= (2 - \tfrac{1}{m})\,\textsc{Opt}
\end{aligned}
$$

Machine Scheduling

Since we have both an upper and lower bound, we have shown the following:

Theorem

The algorithm Ls for minimizing makespan in machine scheduling with $m \geq 1$ machines has competitive ratio $2 - \frac{1}{m}$.

The k-Server Problem [3]

The k-server problem was introduced
in [Manasse, McGeoch, Sleator, STOC, 1988]

The problem concerns the cost of moving servers around in some space.

Thus, we should be precise about what space and costs are.

[3] This exposition is a simplified version of the proofs from [Borodin & El-Yaniv, book, 1998].

Metric Spaces

(M, d) is a *metric space*, if

- M is a set of points
- $d\colon M \times M \to \mathbb{R}$ is a distance function

The distance function should fulfill, for all points $x, y, z \in M$:

- $d(x, x) = 0$
- $x \neq y \Rightarrow d(x, y) > 0$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$ (the triangle inequality)

k-Server: The Problem

SDU

An algorithms controls k mobile servers that are located on points in some metric space, (M, d).

The algorithm must serve some requests. A request, r, is a point in space, and the request is *served* if the algorithm moves a server to that point (or there is a server there already).

Given a sequence of requests, $\sigma = r_1, r_2, \ldots, r_n$, the algorithm must serve each request *sequentially*.

Moving a server from some point r_i to r_j, incurs a *cost* of $d(r_i, r_j)$.

The objective of an algorithm, ALG, is to serve all requests at the smallest possible total cost. We denote ALG's cost on σ by ALG(σ).

In the *online* k-server problem, the algorithm is not informed of the next request (r_{i+1}) until it has served the current (r_i).

k-Server: Laziness

There are no restrictions on how the algorithms move servers. For instance, they may move several servers at the same time just to serve one request. (This may sound silly but we will see later that this can make sense.)

However, for such an algorithm, we can make a *lazy* variant as follows:

Simply run the algorithm *virtually*, i.e., *simulate* the algorithm internally. When a server virtually reaches the request, then actually move that server directly to the request. This could move the server using a different route than what was done virtually, but, by the triangle inequality, the cost is no larger.

Remember where all the servers are in the virtual simulation and continue to the next request.

Due to the triangle inequality, the lazy version of an algorithm will have a total cost less than or equal to the original algorithm.

k-Server: General Lower Bound

SDU

Theorem

Let (M, d) be a metric space with at least $k+1$ points. Then any online algorithm for the k-server problem has a competitive ratio of at least k.

Proof Let ALG be an online algorithm for the k-server problem. We must design an arbitrarily long [4] sequence, σ, such that there exists a constant, b, such that $\text{ALG}(\sigma) \geq k\,\text{OPT}(\sigma) - b$.

Without loss of generality, we assume that ALG is lazy.

Choose any $k+1$ points, $p_1, p_2, \ldots, p_{k+1}$, from M.

Assume ALG initially has its servers on $p_1, \ldots, p_k$.

For any n, we define $= r_1, r_2, \ldots, r_n$ as follows. After serving $i \geq 0$ requests, ALG has left exactly one point, say p_j, uncovered. We define $r_{i+1} = p_j$.

We consider the cost of the sequence $\sigma = r_1, r_2, \ldots, r_{n-1}$.

[4] Actually, it needs to be arbitrarily expensive for OPT, but that is the same thing here.

k-Server: General Lower Bound (2)

Proof (continued)
By definition of σ, ALG serves the request r_i with the server on r_{i+1}.

Thus, the cost of serving the first request is $d(r_2, r_1)$.

Summing up the cost of serving the $n-1$ requests, we obtain the following lower bound on the cost of ALG:

$$\text{ALG}(\sigma) \geq \sum_{i=1}^{n-1} d(r_i, r_{i+1})$$

Next, we need to find an upper bound on the cost of OPT.

k-Server: General Lower Bound (3)

Proof (continued) Let $\mathcal{B}$ be a collection of k different server algorithms defined as follows:

For any set $S \subset \{p_1, p_2, \ldots, p_{k+1}\}$, where $r_1 \in S$ and $|S| = k$, let B_S be the algorithm which starts with a server on all points in S.

Thus, there are k different algorithms in $\mathcal{B}$, depending on which point is initially unoccupied.

On a request, r_i, if r_i is unoccupied, B_S will move the server currently on r_{i-1} to r_i.

k-Server: General Lower Bound (4)

Proof (continued) The k algorithms in $\mathcal{B}$ will remain in different configurations.

We prove this by induction in the number of requests served. It clearly holds after zero requests have been served. For the induction step, consider any two of these algorithms, B_S and $B_{S'}$, and let r_i be the next request.

- r_i is in both configurations: Neither moves and the configurations are still different.
- r_i is in one configuration but not in the other. If r_i is in the configuration, no action is taken, so there is of course still a server on the previous request, r_{i-1}. The other server algorithm will, however, move a server away from r_{i-1} to r_i, so the configurations disagree on r_{i-1}.
- r_i is missing from both configurations. Since, by the induction assumption, the configurations are different, they cannot both be missing r_i, so this case cannot occur.

k-Server: General Lower Bound (5)

Proof (continued) Consider the total cost of all k algorithms at the same time.

Since they are all in different configurations at all times, only one does *not* have a server on any given request r_i, so only *one* algorithm moves from r_{i-1} to r_i. (On the first request, none of them have any cost.)

Thus, the total cost of all the algorithms is at most $\sum_{i=2}^{n-1} d(r_{i-1}, r_i)$.

One of the algorithms, B_S, must have a cost that is at most the average,

$$\frac{1}{k} \sum_{i=2}^{n-1} d(r_{i-1}, r_i)$$

We choose that as our Opt. Clearly, $\text{Alg}(\sigma) \geq k \, \text{Opt}(\sigma)$.

If we would like to assume that Opt starts in the same configuration as Alg, we can set the constant b mentioned first in the proof to the cost of initally moving one server from the point not in S to p_{k+1}. □

k-Server: Algorithms

Now that we know the limits of what we can obtain, how do we design a good algorithm?

Actually, one of the most famous conjectures in the area is the following:

Conjecture

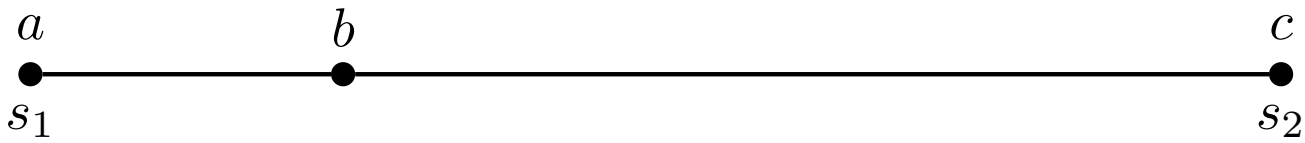
For the k-server problem on any metric space, there exists a deterministic online k-competitive algorithm.

We will not solve that conjecture today; instead we will consider simple metric spaces, where we know how to prove this.

What is the most natural algorithm?

k-Server: The Simplest Nontrivial Problem

Consider the line with three points:



Assume that $k = 2$ and the servers are initially on a and c.

The algorithm Greedy moves the server that can serve the request at as low a cost as possible.

How does it perform on $\sigma = b, a, b, a, b, a, b, a, \ldots$?

How does Opt perform on the same sequence?

We define an algorithm that solves any problem on the line.

k-Server: Double-Coverage on the Real Line

Define algorithm DC as follows:

- If the request is on the same side of all servers, move the nearest server to the request.
- If the request is in between two adjacent servers, move both towards the request at equal speeds until one reaches the request.

How does DC behave on the request from the previous slide?

k-Server: DC's Competitive Ratio

SDU

Theorem

DC is k-competitive.

Proof Let $M_{\min}$ be the minimum weight matching between DC's and Opt's servers. Denote DC's servers by $s_1, s_2, \ldots, s_k$ and also use that notation for their positions. Define $\sum_{\text{DC}} = \sum_{i<j} d(s_i, s_j)$, i.e., all distances between pairs of DC servers.

We assume Opt and DC take turns moving, starting with Opt. We use the potential function

$$k \cdot M_{\min} + \sum_{\text{DC}}$$

Since this is always non-negative, our result follows if we can establish:

1. When Opt moves distance d, the potential increases by at most kd.
2. When DC moves distance d, the potential decreases by at least d.

k-Server: DC's Competitive Ratio (2)

Proof (continued)

Ad 1)

When OPT moves, $\sum_{\text{DC}}$ does not change.

Clearly, if we keep the same matching, the value of that matching will increase by at most d, which happens if OPT's server moves away from the DC server it is matched to. A minimum matching can only be smaller.

k-Server: DC's Competitive Ratio (3)

Proof (continued)

Ad 2)

We consider the two cases of algorithm DC.

Assume only one server moves

This server moves away from all other DC servers, increasing $\sum_{\text{DC}}$ by $(k-1)d$.

OPT has a server on the request, since it moved first. Thus, there exists a minimum weight matching, where DC's moving server is matched to OPT's server on the request. Thus, $M_{\min}$ decreases by at least d.

The decrease in potential is then at least $kd - (k-1)d = d$.

k-Server: DC's Competitive Ratio (4)

Proof (continued)

Assume two servers move

Thus, the servers each move a distance $d/2$.

As in the previous case, one of these two servers can be matched to the request in a minimum weight matching. So, the move by that server decreases $M_{\min}$ by at least $d/2$. The other server may increase that term by $d/2$, but in all, $M_{\min}$ does not increase.

With respect to $\sum_{\mathrm{DC}}$, for any third server, one of the moving servers decrease the distance as much as the other increases the distance. However, the distance between the moving servers is decreased by d.

In total, the potential is decreased by at least d. □

k-Server: Generalizations & Applications

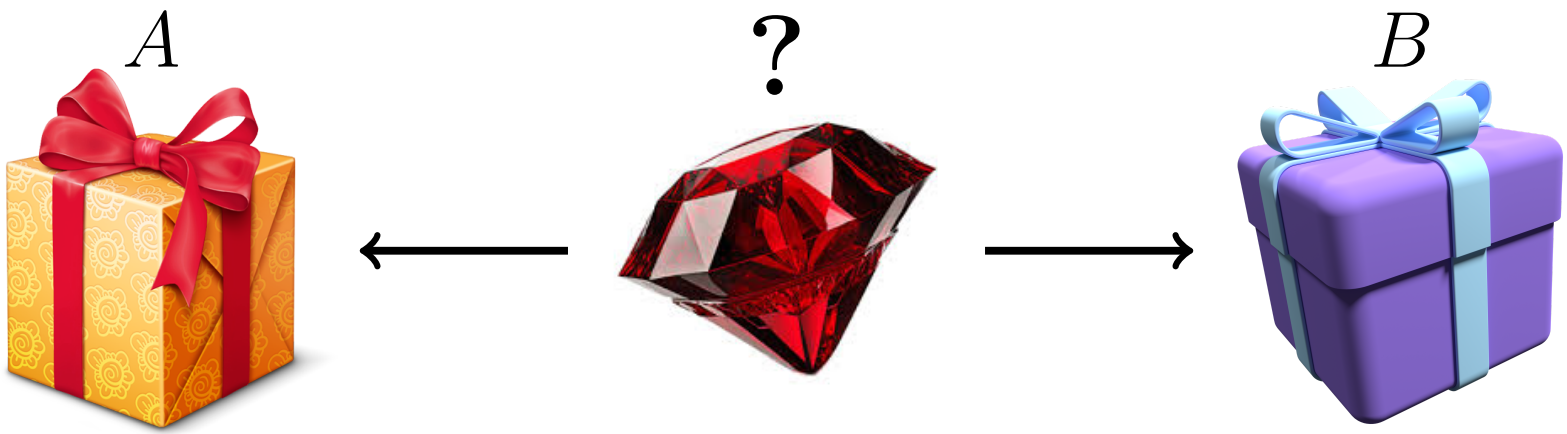
DC generalizes to trees.

More complicated algorithms can perform well on more general metric spaces.

There are obvious applications in the transportation sector.

However, k-server is relevant in many other contexts as well, including paging (as you from operating systems), weighted paging, disk controls, etc.

The Treasure Hunt Problem [5]



- Either box A or B contains an object we need to find.
- The costs of opening A and B are a and b, respectively.

Can we find a good competitive algorithm?

[5] This problem is a simplification of problems such as drilling for resources (minerals, metals, etc.).

The Treasure Hunt Problem (2)

If we search box A first, the adversary will hide it in box B.

So, our cost is always $a + b$, and we get a competitive ratio of $\frac{a+b}{\text{OPT}}$ for some value of OPT.

Thus, we get the best ratio if OPT has a large cost, so we open the cheaper box first (forcing the adversary to hide the treasure in the more expensive box, if the adversary wants to maximize the ratio), and get $\frac{a+b}{\max\{a,b\}}$.

If the boxes are equally expensive to open, this is a ratio of 2.

We will try to do better using randomization...

The Treasure Hunt Problem (3)

SDU

Underlying Assumption

The adversary still knows our algorithm, but not our "coin flip"! [6]

Randomized Algorithm

Let p be the probability that we open A first (we choose p later).

Since our actions depend on probabilities, we consider the *expected* ratios:

Adversary hides object in	Expected competitive ratio
A	$\frac{pa+(1-p)(a+b)}{a}$
B	$\frac{p(a+b)+(1-p)b}{b}$

One of these expressions decreases with p and the other increases. Since the adversary will make it worst possible for us, we should choose p to make these options equally good/bad.

[6] Formally this is referred to as using an *oblivious* adversary.

The Treasure Hunt Problem (4)

$$
\begin{aligned}
\frac{pa+(1-p)(a+b)}{a} &= \frac{p(a+b)+(1-p)b}{b} \\
\Updownarrow & \\
pab + ab + b^2 - pab - pb^2 &= pa^2 + pab + ab - pab \\
\Updownarrow & \\
b^2 - pb^2 &= pa^2 \\
\Updownarrow & \\
b^2 &= p(a^2 + b^2) \\
\Updownarrow & \\
p &= \frac{b^2}{a^2+b^2}
\end{aligned}
$$

The Treasure Hunt Problem (5)

Inserting $p = \frac{b^2}{a^2+b^2}$ into $\frac{pa+(1-p)(a+b)}{a}$, we get the expected competitive ratio

$$\begin{aligned} & \frac{\frac{b^2}{a^2+b^2}a+(1-\frac{b^2}{a^2+b^2})(a+b)}{a} \\ = & \frac{b^2}{a^2+b^2} + \frac{1}{a}\frac{a^2}{a^2+b^2}(a+b) \\ = & \frac{b^2+a(a+b)}{a^2+b^2} \\ = & \frac{a^2+b^2+ab}{a^2+b^2} \end{aligned}$$

Note that this is $\frac{3}{2}$ when $a = b$.

All other situations are better and

$$\lim_{b\to 0+} \frac{a^2+b^2+ab}{a^2+b^2} = \lim_{b\to\infty} \frac{a^2+b^2+ab}{a^2+b^2} = 1$$

References I

Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press (1998)

Mark M. Manasse, Lyle A. McGeoch, and Daniel D. Sleator.

Competitive Algorithms for On-Line Problems.

In *Proc. 20th Annual ACM Symp. on the Theory of Computing*, pages 322–333, 1988.